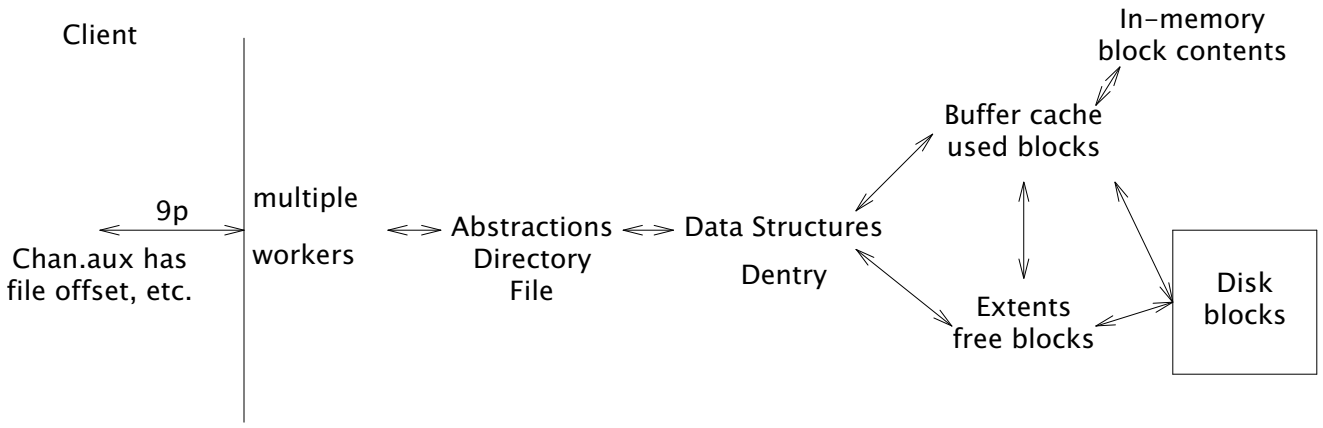
Mafs - Plan 9 userspace file systems

Mafs wants you to be able to understand it, so you can be self-sufficient and fix a crash at two in the morning or satisfy your need for speed or a feature. This empowerment is priceless for those with skin in the game. It provides a reference implementation for creating use-case optimized filesystems.

Mafs is a user space file systems to provide system stability and security. It is based on kfs. It maintains copies of all metadata, creates a copy on any data write and writes to the disk lazily. These equate to greater throughput, the prevention of disk fragmentation, and the display of a recovery path on unsafe shutdowns.

As this document aims to also provide working knowledge, it gratuitously uses the actual commands and the relevant C data structure definitions to convey information.

Mafs Workflow



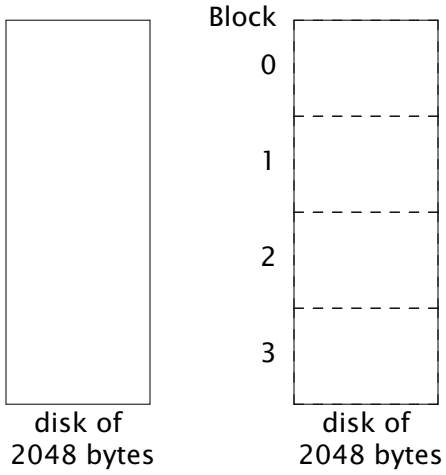
Disk Contents

Mafs organizes and saves content on a disk as directories and files, just like any other filesystem.

The unit of storage is a logical block (not physical sector) of data. Disk space is split into blocks of 512 bytes. A directory entry uses a block and a data block could use upto 2048 blocks (1MiB).

Keeping the directory entries to a block reduces "snowdust" due to the "best fit" algorithm used for assigning disk blocks and memory.

A sample disk of 2048 bytes with 4 blocks.



A block is stored to the disk with a tag in the first byte and the Qid.path in the last 8 bytes. The different types of blocks on a disk are:

```
enum
{
	Tblank = 0,
	Tfree  = 0,	/* free block */
	Tnone  = 0,
	Tdata,		/* actual file contents */
	Tdentry,	/* directory entry, size = Dentrysize */
			/* Tdata & indirect blocks are last, to allow for greater depth */
	Tind0,		/* contains a list of Tdata block numbers for files
				and Tdentry block numbers for directories.*/
	Tind1,		/* contains a list of Tind0 block numbers */
	Tind2,		/* contains a list of Tind1 block numbers */
	Tind3,		/* contains a list of Tind1 block numbers */
	Tind4,		/* contains a list of Tind1 block numbers */
			/* gap for more indirect block depth in future.
				It can be put upto Tind7 without needing any code changes */
	Maxtind,	/* should be Tind0+Niblock */
	MAXTAG = Maxtind,

	Tmaxind = Maxtind - 1,
};
```

Every file or directory is represented on the disk by a directory entry (Dentry). A directory entry uses a block (tag = Tdentry) and is uniquely identifiable by a Qid.

A file's contents are stored in the directory entry itself if they are 320 bytes or lesser. A file stores its contents in blocks with a tag of Tdata if the file size is more than that. A directory holds the directory entries of it's children in blocks each with a tag of Tdentry.

The blocks used by a file or directory entry are listed in their directory entry. As it is not possible to represent big files using the list of blocks available in the directory entry, the blocks are structured to use multiple levels of indirection as the file size increases.

A file's data blocks are identified by a tag of Tdata and that file's Qid.path. A directory's data blocks are identified by a tag of Tdentry and Qid.path of the child directory entry.

(Is this quirky? Should the child's directory entry have a tag with the parent's Qid.path?)

A block number of zero represents the end of the file's contents. If a file is truncated, the data and indirect blocks are given up and the dentry.dblocks[0] = 0.

Mafs does not store the last access time of a file or directory.

A directory entry is defined as:

```
enum {
	Blocksize  = 512ULL,	/* minimum data unit size */

	Maxdatablockunits = 2048,
	Nindperblock= (Blocksize-3*sizeof(u64))/sizeof(u64),/* number of pointers per block */
	Nu64perblock= (Blocksize/sizeof(u64)),	/* number of u64's in a block */
	Dpathidx  = (Blocksize/sizeof(u64) -1),	/* index of path in the last data block, last u64 */

	Namelen = 128,	/* maximum length of a file name, calculated manually */
	Ndblock   = 32,	/* number of direct blocks in a Dentry */
	Niblock   = 5, /* maximum depth of indirect blocks, can increase it to 8 without issues */
};
struct Dentryhdr
{

	u8 tag;
	u8 namelen;
	s16 uid;
	s16 gid;
	s16 muid;	/* 8 */
	u64 size;	/* 0 for directories. For files, size in bytes of the content - 16 */
	u64 pdblkno;	/* block number of the parent directory entry. Will be 0 for root. - 24 */
	u64 pqpath;	/* parent qid.path - 32 */
	u64 mtime;		/* modified time in nano seconds from epoch - 40 */
	u64 qpath;		/* unique identifier Qid.path 48 */
	u32 version;	/* Qid.version 52 */
	u32 mode;		/* same bits as defined in /sys/include/libc.h:/Dir.mode/ - 56 */
	s8 name[Namelen]; /* Namelen = 128 - 184*/
};
struct Datahdr
{
	u8 tag;
	u8 unused;	/* for alignment and future use */
	u16 len;
	u64 dblkno;	/* block number of the directory entry */
};

enum {
	/* size of identifiers used in a Tdata block */
	Ddataidssize = sizeof(Datahdr) +sizeof(u64 /* trailing path */),
	/* max possible size of data that can be stuffed into a Dentry */
	Ddatasize = Blocksize -sizeof(Dentryhdr) -sizeof(u64 /* trailing path */),
	Maxdatablocksize = Maxdatablockunits*Blocksize -Ddataidssize,
};

struct Super
```

```
{
	u64 qidgen;	/* generator for unique ids. starts from 1024. */
	u64 fsok; /* fsck status, using 64 bits to keep it all aligned */
};
struct Dentry
{
	Dentryhdr;
	union
	{
		struct
		{
			u64 dblocks[Ndblock];	/* direct blocks. */
							/* List of Tdata block numbers for files and
							   Tdentry block numbers for directories */
			u64 iblocks[Niblock];	/* indirect blocks */
		};
		Super;

		/* when size <= Dentrysize-184-sizeof(Tag), store the data here itself */
		s8 buf[Ddatasize];
	};
	u64 path; /* same as qid.path */
};
struct Indirect
{
	u8 tagi;	/* the suffix i to avoid union complaining about ambiguous fields */
	u8 pad[7];/* unused, to align to a multiple of 8 */
	u64 dblkno;	/* block number of the directory entry */
	u64 bufa[Nindperblock];
	u64 pathi;/* same as qid.path */
};
struct Data		/* used to unmarshall the disk contents */
{
	Datahdr;
	u8 buf[1]; /* upto Maxdatablocksize, followed by u64 qid.path */
	/* u64 path; same as qid.path at the end of the data content - check consistency */
};
```

A directory entry once assigned is not given up until the parent directory is removed. It is zero'ed if the directory entry is removed. It is reused by the next directory entry created under that parent directory. This removes the need for garbage collection of directory entries on removals and also avoids zero block numbers in the middle of a directory entry's list of blocks. A zero block number while traversing a directory entry's dblocks or iblocks represents the end of directory or file contents. When a directory is removed, the parent will have a directory entry with a tag of Tdentry and Qpnone and the rest of the contents set to zero.

A directory entry is stored in two blocks to have a copy on write. These entries are in consecutive blocks.

A directory's size is always zero.

```
	; tests/6.sizes
	Blocksize 512 Maxdatablockunits 2048
```

Dentryhdr size 184 Ddatasize 320
Dentry size 512 Namelen 128
Datahdr size 12 Ddataidssize 20 Maxdatablocksize 1048556
Namelen 128 Ndblock 32 Niblock 5
Nindperblock 61 Maxdatablocksize 1048556
A Tind0 unit points to 1 data blocks (1048556 bytes)
block points to 61 data blocks
reli start 32 max 92
max size 93*Maxdatablocksize = 97515708 bytes = 92 MiB
A Tind1 unit points to 61 data blocks (63961916 bytes)
block points to 3721 data blocks
reli start 93 max 3813
max size 3814*Maxdatablocksize = 3999192584 bytes = 3 GiB
A Tind2 unit points to 3721 data blocks (3901676876 bytes)
block points to 226981 data blocks
reli start 3814 max 230794
max size 230795*Maxdatablocksize = 242001482020 bytes = 225 GiB
A Tind3 unit points to 226981 data blocks (238002289436 bytes)
block points to 13845841 data blocks
reli start 230795 max 14076635
max size 14076636*Maxdatablocksize = 14760141137616 bytes = 13 TiB
A Tind4 unit points to 13845841 data blocks (14518139655596 bytes)
block points to 844596301 data blocks
reli start 14076636 max 858672936
max size 858672937*Maxdatablocksize = 900366660128972 bytes= 818 TiB

On an empty mafs filesystem mounted at /n/mafs, the disk contents added by the below commands are:
mkdir /n/mafs/dir1
echo test > /n/mafs/dir1/file1

Representation of a file in a directory: /dir1/file1

Block 11 contents: /dir1 Dentry

Tdentry qid.path 64
name dir1
uid 10006
gid -1
muid 10006
size 0
pdblkno 10
pqpath 10
mtime 1653302180819962729
path 64
version 0
mode 20000000777
direct blocks
0 12
1 0
2 0
.
.
.
30 0
31 0
indirect blocks
0 0
1 0
2 0

Block 12 contents: file1 Dentry

Tdentry qid.path 65
name file1
uid 10006
gid -1
muid 10006
size 5
pdblkno 11
pqpath 64
mtime 1653302180823455071
path 65
version 0
mode 666
test

Representation of two files in a directory (/dir2/file1 and /dir2/file2)

Block 13 contents: /dir2 directory entry

Tdentry qid.path 66
name dir2
uid 10006
gid -1
muid 10006
size 0
pdblkno 10
pqpath 10
mtime 1653302180819962729
path 66
version 0
mode 20000000777
direct blocks
0 14
1 15
.
.
.
31 0
indirect blocks
0 0
1 0
2 0

Block 14 contents: file1 directory entry

Tdentry qid.path 67
name file1
uid 10006
gid -1
muid 10006
size 5
pdblkno 13
pqpath 66
mtime 1653302180823455071
path 67
version 0
mode 666
test

iblocks[0] holds the block number of a Tind0 block. A Tind0 block contains a list of Tdata block numbers for files or a list of Tdentry block numbers for directories.

iblocks[1] has the block number of a Tind1 block. A Tind1 block contains a list of Tind0 block numbers.

Similarly, for other iblocks[n] entries, iblocks[n] has the block number of a Tind*n* block. A Tind*n* block contains a list of Tind*(n−1)* block numbers.

Relative index

The zero'th relative index in a directory entry is the first data block. The next relative index is the second data block of the directory entry, and so on.

tests/6.reli shows how a relative index (reli) is translated into an actual disk block number.

To find the actual block number where the first block (zero'th as zero indexed) of a file is stored:

```
tests/6.reli 0 # command, below is the output of this command
reli 0
dblock[0]
```

To find the actual block number where the second block of a file is stored:

tests/6.reli 1
reli 1
dblock[1]

And so on, for the 32nd and 33rd blocks of a file:

tests/6.reli 31
reli 31
dblock[31]

tests/6.reli 32
reli 32
iblock[0] tagstartreli 32
Tind0 reli 0 is at [0] nperindunit 1

Representation of a 2 MiB file (/dir3/2MB.file)

Block 17 contents

Tdentry qid.path 70
name 2MB.file
uid 10006
gid -1
muid 10006
size 2056192
pdblkno 16
pqpath 69
mtime 1653302180819962729
path 70
version 0
mode 664
direct blocks
0 18
1 2066
2 0
.
.
indirect blocks
0 0
1 0
2 0

Block 18 contents

Tdata qid.path 70 dblkno 17 len 2048
0 0123456789

contents of 2MB.file

Representation of a 106490448 bytes file (/big.file)

Block 11 contents

```
Tdentry qid.path 64
name big.file
uid 10006
gid -1
muid 10006
size 26214400
pdblkno 10
pqpath 10
mtime 1653302180819962729
path 64
version 0
mode 666
direct blocks
        0 12
        1 2060
        2 4108
        .
        .
        31 63500
indirect blocks
        0 67596
        1 192525
        2 0
```

Block 12 contents

```
Tdata qid.path 64 dblkno 11 len 2048
        0 0123456789
        .
        .
        .
```

starting contents of big.file

Block 67596 contents

```
Tind0 qid.path 64 dblkno 11
        0 65548
        1 67597
        2 69645
        .
        .
```

Block 65548 contents

```
Tdata qid.path 64 dblkno 11 len 2048
        0123456789
        .
        .
```

more content of big.file

System Files	
Block	Description
0	magic dir entry and data
1	/adm/config dir entry
2	/adm/super dir entry
3	/adm/ dir entry
4	/adm/users/ dir entry
5	/adm/bkp/ dir entry
6	/adm/users/inuse dir entry
7	/adm/frees dir entry
8	/adm/ctl dir entry -- virtual file, empty contents
9	/adm/users/staging dir entry
10	/ direntry

The /adm/ctl file is used to halt or sync the file system. /adm/users is a r/w file that will reload users when written to it. The owner of the /adm/ctl file or any user belonging to the sys group can ream the disk.

There is no /adm/magic directory entry as the block number of the magic block is zero and zero block in a directory entry signifies the end of the directory contents.

Backup blocks

A copy of Config, Super and Root blocks is maintained. This ensures a backup of config, Super and root blocks.

The backup block numbers on the disk are calculated during ream based on the disk size.

Block	Description	Backup Block
1	/adm/config	last block number -1
2	/adm/super	last block number -2
10	/	last block number -3

Mafs needs atleast Nminblocks=14 blocks (7 KiB).

kfs and cwfs use 8192 byte blocks. Hence, they store multiple directory entries (Dentry) per block. They use slot numbers to identify a particular directory entry in a block of directory entries. Mafs avoids that by using 512 byte blocks thus having only one directory entry per block. This avoids locking up other sibling directory entries on access, more compact disk usage and better throughput for large files but also slows down walk()'s as mafs would be using a read() per directory entry.

Users

Users are defined in /adm/users/inuse file. Any changes to it are made through the /adm/users/staging file. All changes are written to the staging file and then inuse file is updated by writing the command users to the /adm/ctl file.

Either all changes to /adm/users/inuse are installed or nothing is installed from the /adm/users/staging file.

The format of /adm/users/inuse is described in users(6).

Buffer cache - Hash buckets with a circular linked list of Iobuf's for collisions.

An Iobuf is used to represent a block in memory. An Iobuf is unique to a block. All disk interaction, except for free block management, happens through an Iobuf. We read a block from the disk into an Iobuf. To update a block on the disk, we write to an Iobuf, which, in-turn gets written to the disk.

An Iobuf is protected by a read-write lock (RWlock). This ensures synchronization across multiple processes updating the same file.

getbuf(), putbuf(), putbufs() and putbuffree() are used to manage Iobuf's. The contents of an Iobuf is not touched unless it is locked by getbuf(). It is unlocked by putbuf(), putbufs() or putbuffree() calls. The Iobuf.dirties Ref is decremented by the mafs writer's dowrite() without a lock(). This is to avoid deadlocks between putbuf() and the writer especially when the writer queue is full.

allocblock() allocates a free block into an Iobuf. allocblocks() allocates a bunch of free blocks with their own Iobuf's.

freeblock() erases the Iobuf and returns the block to the free block management routines.

Iobuf's are organized into a list of hash buckets to speed up access.

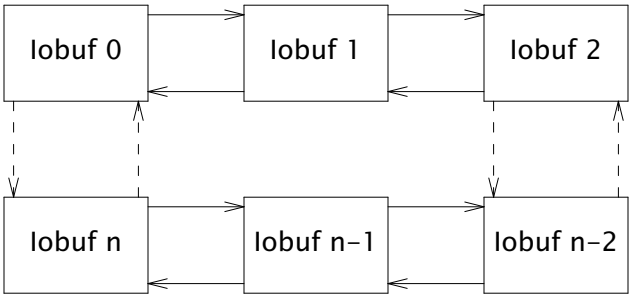
```
Hiob *hiob = nil;     /* array of nbuckets */
struct Hiob           /* Hash bucket */
{
      Iobuf* link;    /* least recently used Iobuf in the circular linked list */
      QLock;          /* controls access to this hash bucket */
      u64  n;         /* count of Iobuf's in the circular list */
};
struct Iobuf
{
      Ref;
      RWLock;               /* controls access to this Iobuf */
      u64  blkno;           /* block number on the disk, primary key */
      u16  len;       /* number of Units */
      Iobuf *fore;    /* for lru */
      Iobuf *back;    /* for lru */
      union{
            u8    *xiobuf;    /* "real" buffer pointer */
            u64  *xiobuf64;
            Data *io;
            Dentry *d;
            Indirect *i;
      };

      u8 *append;           /* appended data added not yet written to disk */
```

```
        u64 appendsize;
        u8 freshalloc;   /* uninitialized blocks on the disk */
        u64 atime;              /* to find old buffers to flush to the disk */
        u8 tag;
        u64 callerpc;    /* when locked, the locker pc for debugging */
};
```

The Iobuf's are arranged into a list of hash buckets. Each bucket points a circular linked list of Iobuf's to handle collisions. If all the Iobuf's in the circular linked list are locked, new Iobuf's are added to this linked list. This circular list is ordered on a least recently used basis. Iobuf's once added to this list are not removed. When an Iobuf is not in the list, the oldest unlocked Iobuf is reused.

Hiob hiob[nbuckets] is a valid representation of the list of hash buckets. The block number is hashed to arrive at the relevant hash bucket index.

hiob[hash(block number)].link = Address of Iobuf0, where Iobuf0 is the least recently used Iobuf.



The size of the buffer cache is: number of hash buckets * collisions per hash bucket * block size. The approximate size of the buffer cache is Nbuckets * Ncollisions * Blocksize * Maxdatablockunits.

When there is an ups, the ideal memnunits (-m) is disk throughput * Nrefresh * ups duration/2. Without an ups, Without an ups, disk throughput * Nrefresh * 3 seems optimal.

If you have RAM to spare, increase Nbuckets instead of Ncollisions as the hash index lookup is faster than searching through a linked list.

Using Maxdatablockunits of 2048 allows bigger data blocks thus providing better throughput for large files but also wastes a lot more memory while writing to small files. If your use case involves requiring more efficient use of memory, setting Maxdatablockunits to 16 keeps the data block sizes down to what cwfs uses.

Iobuf.Ref is used to avoid locking up the hash bucket when a process is waiting for a lock on an Iobuf in that hash bucket.

Iobuf.Ref ensures that an Iobuf is not stolen before another process can get to wlock()'ing it after letting go of the lock on the hash bucket. We cannot hold the lock on the hash bucket until we wlock() the iobuf as that blocks other processes from using the hash bucket. This could also result in a deadlock. For example, the directory entry is block 18, which hashes to a hash index of 7. A writer() locked the directory entry iobuf

and wants to add a data block 84 to the directory entry. Block 84 hashes to the same hash index of 7. Another process wanting to access the directory entry is waiting for a lock on that io buffer. While doing so, it has locked the hash bucket. Now, this has caused a deadlock between both these processes. The first process cannot proceed until it can lock the hash bucket holding block 84 and is still holding the lock on the directory entry in block 18. The second process cannot lock block 18 and is holding the lock on the hash bucket.

```
for locking a buffer:
        qlock(hash bucket); incref(buffer); qunlock(hash bucket);
                wlock(buffer); decref(buffer);

for stealing an unused buffer:
        qlock(hash bucket);
        find a buffer with ref == 0 and wlock()'able.
        qunlock(hash bucket);

for unlocking a buffer:
        wunlock(buffer);
```

Free blocks

Free blocks are managed using Extents. The list of free blocks is stored to the disk when shutting down. If this state is not written, then the file system needs to be checked and the list of free blocks should be updated.

When shutting down, the Extents are written to free blocks. This information can be accessed from /adm/frees. Also, fsok in the super block is set to 1. Mafs does not start until fsok is 1. When fsok = 0, run the sanity check that the unused blocks and the free blocks in /adm/frees match up. disk/reconcile identifies any missing blocks or blocks that are marked as both used and free.

This process of fixing issues and setting fsok to 1 is manual. There is no automatic file system checker as in other file systems. This document aims to empower you with the knowledge to fix your file system issues instead of entrusting your precious data to an arbitrary decision maker such as the file system checker.

A tag of Tfree and Qpnone represent a free block. If a directory entry is removed, the parent will have a zero'ed out child directory entry (Qid.path = 0) and a tag of Tdentry and Qpnone.

Extents

Free blocks and memory are managed using Extents, an abstraction used to manage a continuous list of items.

An Extent represents a continuous list of items. An Extents is a list of such Extent's.

```
struct Extent {
        struct Extent *low, *high; /* sorted by start */
        struct Extent *small, *big;     /* sorted by the number of blocks in this extent */
        u64 start;                      /* where this extent starts from */
        u64 len;                        /* how many units in this extent */
```

```
	/* circular least recently used linked list limited to Nlru items */
	struct Extent *prev, *next;
};
struct Extents {
	Extent *lowest; /* find the first block number in a jiffy */
	QLock lck;
	u64 n;			/* number of extents */
	Rendez isempty; /* fully used, nothing available */

	u8 nlru;		/* number of items in the lru linked list */
	Extent *lru;	/* least recently used extent in the circular lru linked list */
	char name[32];

	void (*flush)(void);
};
```

To allocate n items from Extents, we find the lowest (by block number or memory address) extent that can satisfy our request. If a bigger Extent is available, slice it and take the portion we need.

If there is no available Extent to satisfy our request, panic().

allocblock() and freeblock() use balloc() and bfree() respectively. balloc() assigns blocks from an extent and bfree() adds them to an extent for next allocation.

Extents at memory location 1

lru	100	assuming that the Extent at 100 was used last
el	0	unlocked
n	3	

Extent at 100

blkno	10
len	1
low	0
high	200

Extent at 200

blkno	20
len	3
low	100
high	300

Extent at 300

blkno	30
len	2
low	200
high	0

\+ freed block numbers 11,12,13,14 =

Extent at 100

blkno	10
len	5
low	0
high	200

Extent at 200

blkno	20
len	3
low	100
high	300

Extent at 300

blkno	30
len	2
low	200
high	0

Extents before

blkno	len
20	3

\+ Block number 40 followed by 3 free blocks =

Extents after

blkno	len
20	3
40	4

Extents before

blkno	len
100	5
110	3

\+ Block number 105 followed by 4 free blocks =

Extents after

blkno	len
100	13

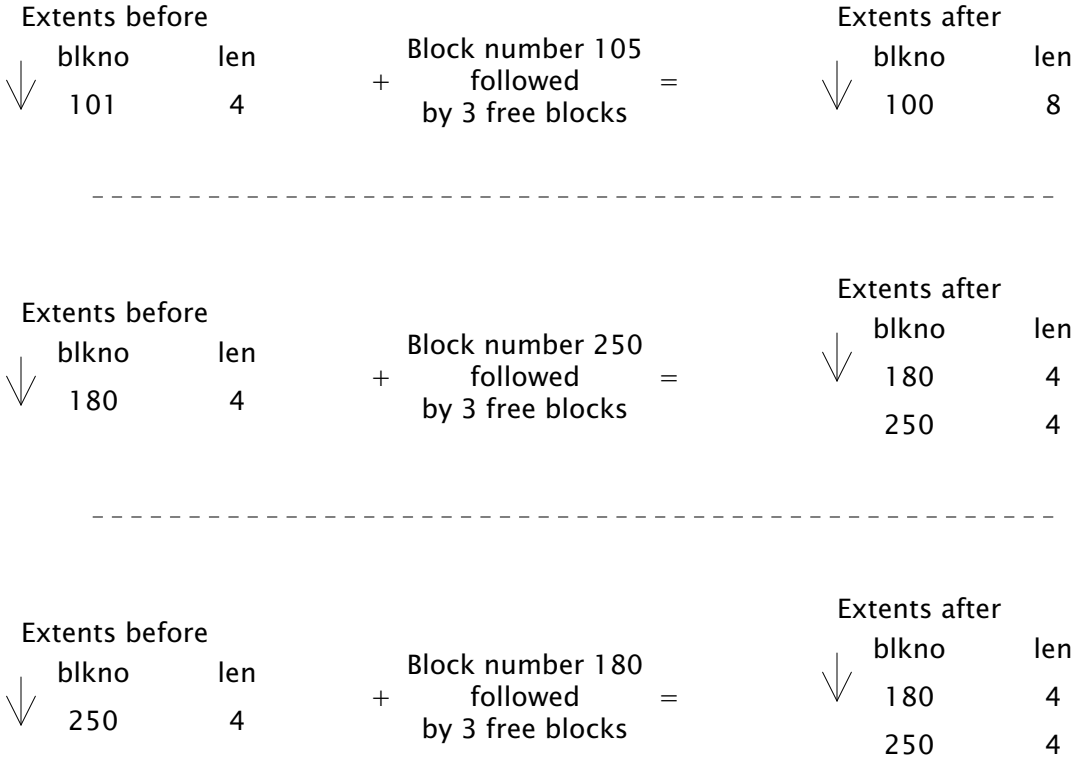
Extents before

blkno	len
105	4

\+ Block number 101 followed by 3 free blocks =

Extents after

blkno	len
101	8



Kfs stores the list of free blocks in a Tfrees block and the Superblock. Instead we use block management routines, similar to pool.h, to allocate and monitor free blocks. On shutdown(), the block management routines (extents.[ch]) store state into the free blocks. This can be read from /adm/frees. On startup, this is read back by the block management routines. On a crash, the fsck can walk the directory structure to identify the free blocks and recreate /adm/frees.

Code details

Program	Description
disk/mafs	Start mafs on a disk.
disk/fsck	Check file system on an improper shutdown.
disk/free	List the free blocks. It reads the contents of /adm/frees.
disk/used	List the used blocks by traversing all directory entries.
disk/block	Show the contents of a block.
disk/unused	Lists the unused blocks when given extents of used blocks.
disk/updatefrees	Update the contents of /adm/frees.

File	Description	chatty9p
9p.c	9p transactions	2
blk.c	routines to show blocks.	
console.c	obsolete. /adm/ctl is the console.	
ctl.c	/adm/ctl operations.	
dentry.c	encode/decode the file system abstraction into block operations.	3
extents.[ch]	routines to manage the free blocks.	6
iobuf.c	routines on Iobuf's. The bkp() routines operate on Iobuf's.	5
sub.c	initialization and super block related routines.	2
tag.c	routines to manage a relative index (reli) in a directory entry.	
user.c	user management routines.	

A Chan's state could get out of sync with the contents if another process changes the on-disk state. Ephase error occurs when that happens.

For throughput, multiple processes are used to service 9p i/o requests when the -s flag is not used.

Useful commands:

Ream and start single process Mafs on a disk and also mount it for use.

```
mount -c <{disk/mafs -s -r mafs_myservice mydisk <[0=1]} /n/mafs_myservice
    -s: use stdin and stdout for communication
    -r mafs_myservice: ream the disk using mafs_myservice as the service name
     mydisk: running mafs on the disk, mydisk
```

Ream and start multiple-process mafs on a disk.

```
disk/mafs -r mafs_myservice mydisk
mount -c /srv/mafs_myservice /n/mafs_myservice
```

Ream and start mafs on a file. Also, mount thet filesystem at /n/mafs_myservice.

```
dd -if /dev/zero -of myfile -bs 512 -count 128 # 64KB file
mount -c <{disk/mafs -s -r mafs_service myfile <[0=1]} /n/mafs_myservice

# to reuse the contents of myfile later, remove -r (ream) from the above command.
mount -c <{disk/mafs -s myfile <[0=1]} /n/mafs_myservice
```

Prepare and use a disk (/dev/sdF1) for mafs.

```
disk/fdisk -bawp /dev/sdF1/data   # partition the disk
echo '
a fs 9  $-7
w
p
q' | disk/prep -b  /dev/sdF1/plan9 # add an fs plan 9 partition to the disk
disk/mafs -r mafs_sdF1 /dev/sdF1/fs     # -r to ream the disk
mount -c /srv/mafs_sdF1 /n/mafs_sdF1
```

```
# for using the mafs file system on the disk later on
disk/mafs  /dev/sdF1/fs # no -r
mount -c /srv/mafs_sdF1 /n/mafs_sdF1
```

Starting mafs on a 100 MiB file. The below commands create a disk.file to use as a disk. Mount /n/mafs_disk.file for the file system.

```
dd -if /dev/zero -of disk.file -bs 1m -count 100;
mount -c <{disk/mafs -s -r mafs_disk.file disk.file <[0=1]} /n/mafs_disk.file
```

Starting mafs on a RAM file. The below commands create a ramfs filesystem to use as a disk.

```
ramfs -m /n/mafs_ramfs
touch /n/mafs_ramfs/file
dd -if /dev/zero -of /n/mafs_ramfs/file -count 100 -bs 1m
disk/mafs -r mafs_ramfs_file /n/mafs_ramfs/file
mount -c /srv/mafs_ramfs_file /n/mafs_ramfs_file
```

Sync Mafs. This command does not return until all the writes are written to the disk. So, could take a long time if you have a long writer queue.

```
echo sync >> /n/mafs_myservice/adm/ctl
```

Stop Mafs: There are 2 ways to shutdown:

1. Unmount and remove the /srv/mfs_service file (can be rm and unmount too).
2. Write halt into the /adm/ctl file. Unmount() the mafs file system to keep it clean.

In the first instance, the srv() process is driving the shutdown. It calls fsend(). rm /srv/mfs_service file does not wait for fsend() to finish. Hence, there is no way to ensure that memory contents have been flushed to the disk. If the system is shutdown or restarted immediately, there is a very high possibility that the filesystem will be in an inconsistent state.

In the second instance, fsend() is called by the worker process. It does not return until all the pending writes have been flushed to the disk. It also removes the /srv/mafs_service file and also stops the srv() process. Hence, this is the preferred approach to shutting down the file system.

There is no way to unmount() automatically on shutdown. The mount() and unmount() calls are client driven and it is not the responsibility of the server to find all the clients that mounted it. Just shutdown and let the respective clients deal with their mess.

The below command does not return until all the writes are written to the disk. So, could take a long time if you have a long writer queue. This is the proper way to shutdown the mafs file system.

```
echo halt >> /n/mafs_myservice/adm/ctl
```

Interpret the contents of a block based on the tag and write out a single formatted block based on the tag

```
disk/block  tests/test.0/disk 22
```

Traverse the directory heirarchy and write out all the used block numbers. disk/reconcile uses the output of this to reconcile the list of used blocks with the list of free blocks. Also, writes the invalid blocks to stderr. Starting from root, walk down each directory entry printing out the linked blocks with invalid tags. (Why not just write out the list of dirty blocks too? instead of using a different command for it?)

```
disk/used  tests/test.0/disk
```

From the contents of /adm/frees show the list of free blocks. disk/reconcile uses the output of this to reconcile the list of used blocks with the list of free blocks.

```
disk/free  tests/test.0/disk
```

Read two lists of block numbers and flag the common and missing blocks.

```
disk/reconcile -u <{disk/used tests/test.0/disk} \
      -F <{disk/free  tests/test.0/disk} 32
```

Find traverses the directory heirarchy and identifies the file that a block number belongs to.

```
disk/find tests/test.0/disk 17
```

Find the total number of blocks on a disk.

```
dd -if /dev/sdF1/fs -bs 512 -iseek 2 -count 1 -quiet 1 | awk '$1 == "nblocks" { print $2 }'

disk/block /dev/sdF1/fs 2 | awk '$1 == "nblocks" { print $2 }'
```

Build the list of free blocks. This should match the contents of /adm/frees.

```
disk/unused 11721040049 <{disk/used /dev/sdF1/fs} # 11721040049 = total number of disk blocks
disk/unused 32 <{disk/used test.0/disk} # 32 = total number of disk blocks
```

Change the contents of /adm/frees.

```
disk/updatefrees tests/test.0/disk <{disk/unused 32 <{disk/used tests/test.0/disk}}
disk/updatefrees /dev/sdF1/fs <{disk/unused 11721040049 <{disk/used /dev/sdF1/fs}}
```

A sanity check that the file system is not corrupt by comparing that the unused blocks and free blocks match up. $nblocks is the total number of disk blocks. $disk is the disk.

```
diff <{disk/unused -l 32 <{disk/used tests/test.0/disk}}} <{disk/free tests/test.0/disk}
```

Check and correct a crashed filesystem.

```
disk=/dev/sdF1/fs
nblocks=`{disk/block /dev/sdF1/fs 2 | awk '$1 == "nblocks"{ print $2 }'}
disk/updatefrees /dev/sdF1/fs <{disk/unused $nblocks <{disk/used /dev/sdF1/fs}
disk/fsok /dev/sdF1/fs

disk=/dev/sdF1/fs
nblocks=`{disk/block $disk 2 | awk '$1 == "nblocks"{ print $2 }'}
disk/updatefrees $disk <{disk/unused $nblocks <{disk/used $disk}
```

```
disk/fsok $disk
```

Tests

Program	Description
tests/regress.rc	All regression tests
tests/chkextents.rc	Unit tests on extents
tests/chkreli.rc	Unit tests on relative index lookups
tests/chknlastdatablocks.rc	Unit tests on the number of blocks in the last Tdata block
tests/6.offsets	Write file using different offsets to test mafswrite()
tests/6.sizes	Show the effects of the different parameters
tests/6.testextents	Test extents.[ch] state changes
tests/6.reli	Translate relative index to block number on a disk

The below disk state tests:
1. Initialize a disk for mafs.
2. Run mafs on that dsk.
3. Stop mafs.
4. Compare the contents with the expected contents (tests/test.0/blocks/*).

Disk State	
Test	Description
tests/test.0	empty disk
tests/test.1	create a file /dir1/file1 and echo test into it
tests/test.2	writes at different offsets to a file and then removes the file
tests/test.3	write, read and delete files with sizes upto 16384 blocks
tests/test.4	directory copy
tests/test.5	fcp gzipped files
tests/test.6	df
tests/test.7	multiple processes working on the filesystem simultaneously
tests/test.8	check backup blocks locations
tests/test.9	examples used by this document
tests/test.a	write, read and delete a 100MB file
tests/test.b	duplicate of test.2 but seeded with random data
tests/test.d	seed with random data and do mkdir -p a/b/c/d/e/f/g/h
tests/test.e	seed with random data and test directory and file deletions
tests/test.f	test restart

Extents behaviour	
Test	Description
tests/extents/addabove	Figure 1 of the Extents section
tests/extents/addabove1	Figure 2 of the Extents section
tests/extents/addbelow	Figure 3 of the Extents section
tests/extents/mergeabove	Figure 4 of the Extents section
tests/extents/mergenext	Figure 5 of the Extents section
tests/extents/mergeprevious	Figure 6 of the Extents section

To run all the regression tests:

```
cd tests/; ./regress.rc
```

To loop through all the blocks of a test:

```
for(t in tests/test.2/blocks/^‘{seq 0 39}*){ echo $t; echo ’---------’; cat $t; echo }
```

Performance metrics

```
ramfs -m /n/ramfs
touch /n/ramfs/file
cat /dev/zero | tput -p > /n/ramfs/file
        196.00 MB/s
        198.76 MB/s
        187.58 MB/s
        176.96 MB/s
        175.87 MB/s
        180.42 MB/s
        183.52 MB/s
        185.99 MB/s
        187.96 MB/s
        189.54 MB/s
        190.83 MB/s
        191.89 MB/s
        192.80 MB/s
dd -if /dev/zero -of /n/ramfs/file -count 700 -bs 1m

disk/mafs -r mafs_ramfs_file /n/ramfs/file
mount -c /srv/mafs_ramfs_file /n/mafs_ramfs_file
cat /dev/zero | tput -p > /n/mafs_ramfs_file/zeros.file # increase memunits for speed
        122.50 MB/s
        122.13 MB/s
        122.27 MB/s
        122.28 MB/s
echo halt >> /n/mafs_ramfs_file/adm/ctl; lc /srv
unmount /n/mafs_ramfs_file

dd -if /dev/zero -of /n/ramfs/file -count 700 -bs 1m
hjfs -f /n/ramfs/file -r
echo allow >>/srv/hjfs.cmd
mount -c /srv/hjfs /n/hjfs/
cat /dev/zero | tput -p > /n/hjfs/zeros.file
        70.85 MB/s
        71.02 MB/s
        70.88 MB/s
        70.62 MB/s
        70.46 MB/s
        70.54 MB/s
        69.74 MB/s
        68.63 MB/s
        67.74 MB/s
        67.02 MB/s
        63.81 MB/s
echo halt >>/srv/hjfs.cmd
unmount /n/ramfs

time disk/used /dev/sdF1/fs > /tmp/used.blocks # on 600 GiB data stored to a SATA disk
```

```
3.45u 30.41s 3201.71r    disk/used /dev/sdF1/fs

ramfs -m /n/ramfs
touch /n/ramfs/file
dd -if /dev/zero -of /n/ramfs/file -count 1k -bs 2m

disk/mafs -r mafs_ramfs_file /n/ramfs/file
mount -c /srv/mafs_ramfs_file /n/mafs_ramfs_file
cd /n/mafs_ramfs_file
time git/clone /dist/plan9front
time walk plan9front >/dev/null
time cat '{walk plan9front}>/dev/null
echo halt >> /n/mafs_ramfs_file/adm/ctl; lc /srv
unmount /n/mafs_ramfs_file

hjfs -f /n/ramfs/file -r
echo allow >>/srv/hjfs.cmd
mount -c /srv/hjfs /n/hjfs/
cd /n/hjfs
time git/clone /dist/plan9front
time walk plan9front >/dev/null
time cat '{walk plan9front}>/dev/null
echo halt >>/srv/hjfs.cmd
unmount /n/ramfs
```

Profiling instructions:

```
Set LDFLAGS=-p in the mkfile and install the executables.
profilesize=2000000
ramfs -m /n/ramfs
touch /n/ramfs/file
dd -if /dev/zero -of /n/ramfs/file -count 700 -bs 1m
mount -c <{disk/mafs -s -r mafs_disk.file /n/ramfs/file <[0=1]} /n/mafs_ramfs_file
cat /dev/zero | tput -p > /n/mafs_ramfs_file/zeros.file
        57.94 MB/s
        55.27 MB/s
        46.67 MB/s
echo halt >> /n/mafs_ramfs_file/adm/ctl; lc /srv
```

Limitations

As we use packed structs to store data to the disk, a disk with mafs is not portable to a machine using a different endian system.

Design considerations

For exclusive use (mode has DMEXCL bit set) files, there is no timeout.

Use an fs(3) device for RAID or other configuration involving multiple disks.

Why are you not using a checksum to verify the contents?
Checksums are probabilistic and can be implemented as a bespoke application instead

of complicating the file system implementation.

Source

http://git.9front.org/plan9front/mafs/HEAD/info.html

References

[1] Sean Quinlan, "A Cached WORM File System," Software--Practice and Experience, Vol 21., No 12., December 1991, pp. 1289-1299
[2] Ken Thompson, Geoff Collyer, "The 64-bit Standalone Plan 9 File Server"